Linux 0.11 文件系统的实现

第一节. 概述

Linux 0.11 只支持一种文件系统，那就是 Minux 1.0 文件系统。这是一个与 Minix 操作系统相兼容，并且与 Unix 文件系统较为接近的索引文件系统。

第二节. 静态结构

Minux 1.0 文件系统一共由六个部分组成

1. 引导块，俗称 MBR，是磁盘的第一个盘块，用来标记并在系统开机时 BIOS 自检完毕后，被自动读入到内存 0x7c00 位置运行。它的主要任务是提供磁盘分区信息和最早的操作系统引导工作。它的大小只有 0x0200 字节（在 Minix 1.0 文件系统中，它实际占 1 个盘块，大小 1K），并且在第 0x01FE 的位置必须填充 0x55AA 来告之 BIOS 该磁盘可以被引导。引导块的结构是由硬件规定的，现在普遍使用的引导块结构是 IBM 兼容 PC 采用的结构。

Structure of a master boot record

Address			Description		Size in bytes
Hex	Oct	Dec			
0000	0000	0	code area		**440** (max. 446)
01B8	0670	440	disk signature (optional)		**4**
01BC	0674	444	Usually nulls; 0x0000		**2**
01BE	0676	446	**Table of primary partitions** (Four 16-byte entries, IBM partition table scheme)		**64**
01FE	0776	510	55h	MBR signature; 0xAA55	**2**
01FF	0777	511	AAh		
MBR, total size: 446 + 64 + 2 =					**512**

2. 超级块，提供磁盘上的文件系统结构信息。例如文件系统的幻数，最大文件长度，inode 结点数和逻辑块数等等。超级块既静态驻扎在磁盘第二个盘块上，又会在磁盘被挂载时驻扎进内存（所谓挂载，主要任务就是载入磁盘的超级块和 inode 位图和逻辑块位图）。不过在内存中的超级块结构更加庞大，增加了很多运行时的参数，例如于 inode 和逻辑块的位图在高速缓冲区的位置（在 Minix1.0 文件系统中高速缓冲区中能容纳 8 个 inode 位图和 8 个逻辑块位图），设备号，磁盘挂载到的 inode 结点等信息。Linux 0.11 在内存中为超级块开辟的空间只能容纳 8 个超级块（定义在 include/linux/fs.h），也就

是说 Linux 0.11 只支持 8 个磁盘同时挂载。

3. inode 位图。所谓 inode，是一种记录文件或目录所有源信息的数据结构，驻扎在磁盘和内存中，它记录的信息有文件类型（宏定义在 include/const.h 内），文件权限，文件拥有主和拥有组，最近修改时间，被硬链接的次数，以及最重要的，文件占用的所有逻辑块的编号，在 Minix 1.0 文件系统中一个文件最多占用 7+512+512×512 个逻辑块，如果是在内存中，它还会记录有等待该 inode 解锁的进程链表，所在的设备号，引用计数，是否锁定，是否是脏数据块（所谓脏数据块，是指当 inode 被载入到内存后被修改过，但修改后的 inode 还没有写回磁盘的状态），是否安装其他文件系统，文件读写指针，是否是最新内容。它的标识符是一个数字，通常被存为 i 结点号。（你也许会很惊讶，inode 从不记录文件名和文件路径，这得益于在索引文件系统中，硬链接的存在，使得一个文件可能拥有多个文件名和多个路径。因此，文件名由文件所在目录决定，文件路径则在文件打开时根据当前用户的所在路径）所谓 inode 位图，正是标记哪些 i 结点号已经被使用过，哪些还没有被使用过的一张位图表。0 号 inode 不被使用。
4. 逻辑块位图。与 inode 位图类似。它记录磁盘哪些逻辑块已经被使用过，哪些没有。在 Minix 1.0 文件系统中，逻辑块大小与盘块大小相同，都是 1K，但是逻辑块是从文件系统第一个数据区的盘块开始编号的。
5. 数据区，占满除上述区域以外的磁盘所有区域。

第三节. 高速缓冲区

讲 Minix 1.0 文件系统，就必须先从高速缓冲区讲起。这是因为高速缓冲区是内存和磁盘之间进行数据传输的桥梁。在 Linux 0.11 系统中，绝不可能实现内存与磁盘的直接数据交互，因为没有这些代码。内存与磁盘都只和高速缓冲区打交道，即使是磁盘上的超级块，位图等特殊块都不例外，因此，不能理解高速缓冲区就不可能理解文件系统。

在 Linux 0.11 中，高速缓冲区位于内核和主内存之间，并且至少拥有从内核代码到内存 640K 之间的区域。如果内存较大，则还会从内存 1M 的区域开始扩张。如果内存超过 12MB，缓冲区末端位置在 4M，否则如果超过 6MB，缓冲区末端位置在 2MB。

高速缓冲区虽然位于内存，但不使用段页式管理，而是用和盘块，逻辑块大小相同（1K）的缓冲块管理。并在高速缓冲区前部创建缓冲头（buffer_head）——一种记录缓冲块原信息的数据结构，每个缓冲头对应一个缓冲块。缓冲头使用哈希散列和队列两种方式管理，当缓冲块中存储了某个设备中某一盘块的信息时，缓冲头会根据设备号和盘块号来决定自己被连入哈希散列的某个区域。在 Linux 0.11 系统中，似乎所有的缓冲块都是空闲的，都在空闲队列中，即使缓冲块是脏块，或是引用计数大于 0 甚至它还处于被锁定状态（似乎和 Unix 并不相同）。每当需要新的缓冲块时，就从空闲队列中取出一个相对空闲的缓冲块，这个缓冲块也许是脏块，务必将它写回磁

盘，否则数据将被覆盖。这个缓冲块也许引用计数大于 0，请务必耐心等待直到彻底释放。然后修改缓冲头在哈希散列中的位置，使它永远与设备号和盘块号对应。这里的情况就请想象如果你去一家餐馆，而餐馆没有空位了，你会怎么办？只能找一桌看上去快吃完的然后等在他的旁边吧。这里的算法也很类似，但是由于多任务的存在，还会更复杂些，详细的机制请看下文叙述。

正如前面所言，高速缓冲区的主要任务是架起内存和磁盘之间进行数据传输的桥梁。当从磁盘上读取盘块后，总是将盘块写入高速缓冲区，然后内存从高速缓冲区中读取盘块数据（前面已经说过，这个步骤不能避免）。以后，如果再读取这个盘块的信息，只要这个缓冲块没有被换出，就不再访问磁盘。如果要对这个盘块进行写入，也只写入缓冲区，并标记它为脏块，没有磁盘的事。只有当发生如空闲块是脏块的情况，或是调用 sync 系统调用，才将缓冲块写回磁盘。我们都知道访问磁盘和访问内存的速度相差甚远，因此高速缓冲区对加速文件系统效果显著。据我所知，Windows 操作系统没有实现高速缓冲区，Windows 应用程序通过系统调用直接访问磁盘，这使得对磁盘的访问非常频繁，影响了程序的执行速度，同时，也增加了对磁盘的损耗。我们都知道，很多 Windows 平台的下载软件，比如迅雷或是 Flashget，都实现了自己的缓冲区，以保护磁盘，而 Linux 平台的下载软件，比如 Transmission，则不需要这么做，因为操作系统本身已经实现了这个功能。

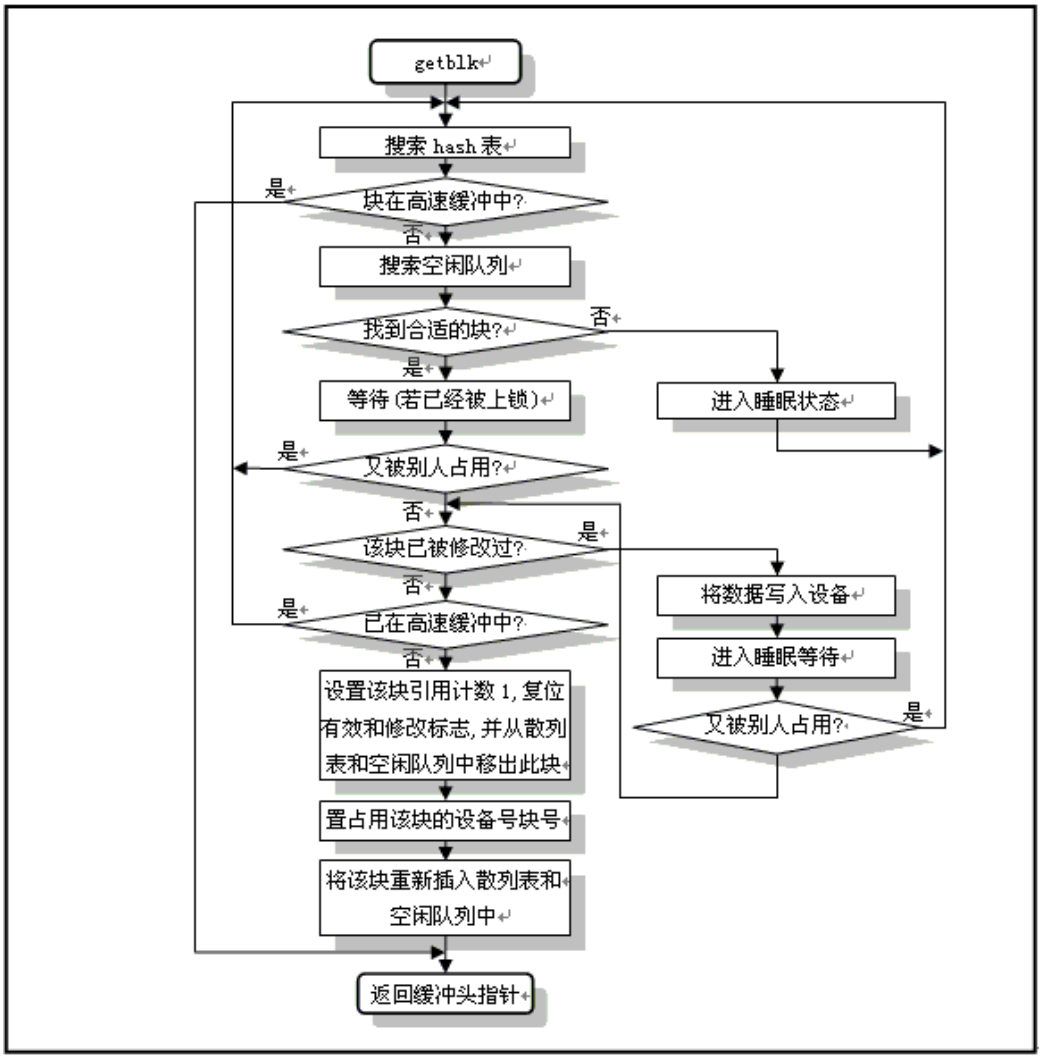
当然，高速缓冲区并非完全都是优点，它的缺陷在于一旦磁盘被强行拔出，或是计算机突然断电，缓冲区中的脏块没有被写入磁盘，那么会造成磁盘的数据丢失，甚至造成磁盘上的文件系统损坏。对于被强行拔出，Linux 的作者 Linus 在注释中写到，People changing disketees in the middle of an operation deserve to loose，在操作过程中更换磁盘从而丢失数据的用户是咎由自取。不过，现代 Linux 采用的文件系统都是日志式文件系统，这种缺陷因此被极大程度的弥补了。

下面将对高速缓冲区的主要实现，fs/buffer.c 文件的几个实现主要功能的函数的算法进行论述。

buffer_init() 初始化缓冲区。首先，确定缓冲区的区域，缓冲区的起始位置就在内核代码的末端，当内核在编译的时候，System.map 文件会记录内核代码的末端位置，当编译到 buffer.c 的时候，ld 会将这个数据链接给 buffer.c，这样缓冲区起始位置就确定了。缓冲区的末端位置则在 boot/head.s 的时候初步决定，这里在做一次修改，因此 head.s 中的逻辑是，如果内存没有大于 6M，缓冲区的末端位置在 1MB，这里就判定当缓冲区末端位置在 1MB 的时候，将其设置为 640K。然后正向初始化缓冲头，将所有数据都设置为 0，除了指向对应缓冲块的指针，特别注意，第一个缓冲头指向最后一个缓冲块，第二个缓冲头指向倒数第二个缓冲块，以此类推。将所有缓冲头放入空闲队列，队头指向第一个缓冲头，并使队列循环。注意，千万不能碰到内存 640K~1M 的区域，那里归显存和 BIOS ROM 所有。最后初始化哈希散列，将散列表每一项的指针都设置为 NULL 即可。

getblk() 获取指定设备号和盘块号的缓冲块，返回指向缓冲头的指针。这个函数做的事情并不多，但由于多任务的存在，竞争条件导致这个函数算法异常复杂。首先，将根据指定的设备号和盘块号在哈希散列中搜索缓冲头，如果存在，返回缓冲头即可。（这里描述下 Linux 0.11 哈希散列，在 Linux 0.11 中，哈希散列由 307 个缓冲头链表组成，初始情况下都是空链表，但是当缓冲块被某设备的某盘块使用，根据它的设备号和盘块号，以及哈希算法计算出的哈希值将决定对应的缓冲头被链入 307 个缓冲头链表中的哪一个。哈希算法是(设备号^盘块号)

%307，相同哈希值的缓冲头将被链入同一链表。当搜索缓冲块时，用同样的算法算出哈希值，然后找到链表，遍历链表直到找到缓冲块，或是找不到返回 NULL。）如果找不到缓冲头，将搜索空闲队列，取一个较为空闲的缓冲块，这里所说的较为空闲是指引用计数等于 0，并且既不是脏块又不是被锁定的块。如果没有，找引用计数为 0 却处于被锁定状态的干净的缓冲块（这种现象通常是缓冲块内容正在被写回，同时它的主人已经释放了对它的控制），这种块的释放速度很快，不超过一次硬盘写回的时间，可以等待。如果没有，找引用计数为 0，没有被锁定的脏缓冲块，这种块意味着你必须先写回再使用，等待时间正好是一次硬盘写回的时间。如果没有，找引用计数为 0，被锁定的脏缓冲块（真有这种块吗？）。如果还是没有，证明引用计数为 0 的缓冲块已经不存在了，没有必要再妥协，可以睡眠等待其他进程唤醒，然后重新再找。假设已经找到，如果是脏块，写回，如果锁定，等待解锁。这两个动作都需要一定时间，因此都会睡眠，必须在睡眠后重新判定找到的块是否被其他进程占用，如果被占用，重新搜索。这里的情况就想象如果你找女朋友，你看上一个美女，并确定她还没有男朋友，你苦思冥想一夜终于写出一封情书，正当你想把情书交给美女时，却发现一夜之间她有了男朋友，那么你只能重新再找了。类似的情况在后面几个章节中会多次发生，必须理解。然后由于睡眠过，甚至无法确定有没有另一个进程在这段时间成功获取了和你一样的缓冲块。绝不能存在两块设备号和盘块号完全一致的缓冲块，否则会出现问题。不得不释放当前缓冲块，然后重新再找。如果这种情况没有发生，那么可以确定找到的缓冲块没有问题，将缓冲块进行设置，重新插入空闲队列和哈希散列，然后返回指向缓冲头的指针。



bread()和 brelse() 分别是获取缓冲块和释放缓冲块的函数。bread()主要封装了 getblk()函数，并在它的行为上再增加将指定设备上的指定盘块读入缓冲块缓冲区的功能，使它真正实现了获取指定设备上指定盘块的缓冲块的功能。brelse()的实现很简单，首先等待缓冲块解锁，然后引用计数减 1，唤醒之前由于找不到引用计数为 0 的缓冲块而睡眠的进程。

第四节. 位图管理

可能你已经注意到，Minix 1.0 文件系统共有两种位图，一种是 inode 位图，另一种是逻辑块位图，两种位图非常相似，都是用 1 位来表示某一 i 结点号或逻辑块号是否被使用，如果没有被使用，置 0，如果被使用，置 1，如果不存在，也同样置 1。在 Linux 中，Linus 使用汇编语言来扫描位图，以用来搜索或是设置，这是因为 C 语言关于位操作的功能非常弱而且低效，使用汇编指令则高效很多。位图管理的实现写在 fs/bitmap.c 文件中，没有任何复杂算法，这里只用逻辑块为例做简单叙

述，至于 inode，则与之类似。

new_block() 在指定设备的超级块中的逻辑块位图所在的缓冲块中（注意，只要被挂载，逻辑块位图和 inode 位图都进入了缓冲区，且不会被换出）找到一个 0 位，如果找不到将返回 0，逻辑块的 0 无效，所以 0 号逻辑块总是空块。将找到的 0 位置位为 1，将该缓冲块标记为脏块。现在我们已经拥有一个可用的逻辑块号，调用 getblk()（不是 bread()，因为获得的逻辑块是空块）获取指定的缓冲块，删除缓冲块数据，标记它为最新和脏块。

free_block() 释放指定设备上指定盘块的缓冲块，先找到目标缓冲块，它一定存在于高速缓冲区中，然后查看它的引用计数，如果大于 1，证明还有进程在使用这个缓冲块，不能释放，仅仅是引用计数减 1。否则，复位它的最新标记和脏块标记，这会使得缓冲块上的数据无效，因为连盘块都无效了，缓冲块的数据当然也毫无意义了。然后删除缓冲块，将超级块中的逻辑块位图所在的缓冲块中的指定为复位，标记这个缓冲块为脏块，即可。

第五节. 多级索引式逻辑块管理

Minix 1.0 文件系统作为一个索引文件系统，具有索引文件系统的最大特点，多级索引式逻辑块管理。所谓索引式逻辑块管理，就是在文件的 inode 中，记录这个文件的所有逻辑块号（被称为直接块），所谓多级，是指如果文件较大，inode 已经无法记录所有逻辑块号，将申请空间来存放多余的逻辑块号，并在 inode 中记录这片空间的首地址（被称为一级索引式）。如果依旧不够，申请更多的空间来记录逻辑块号（此时，这些空间被称为二级间接块），然后再申请一片空间（这片空间被称为一级间接块）来记录这些空间的首地址，在 inode 中记录这个空间（指一级间接块）的首地址（被称为二级索引式），这样的机制就叫多级索引式逻辑块管理。不过在 Minix 1.0 文件系统中，最多只用了两级。其中第 0~第 6 个逻辑块直接记录在 inode 中，第 7~第 518 块按一级索引式管理的方法，记录在一级间接块中，第 519~第 262662 块按二级索引式管理的方法，记录在二级索引块中。无论是一级间接块还是二级间接块，无论记录的是逻辑块号还是下一级的间接块地址，总是 1K 大小，512 条记录，每条记录占 2 个字节（2 个字节最多表示 65536 个 inode）。每当读取文件时，就按照这个顺序将逻辑块一个一个读入高速缓冲区，这样，整个文件就被读入。删除文件时，就将所有的直接块，一级间接块和二级间接块的逻辑块号在逻辑块位图中复位，然后清除 inode 中的相关记录即可。

第六节. inode

inode 是索引式文件系统的特色，通过它可以确定一个唯一的文件并记录它的源信息。在索引式文

件系统中，你在目录项中所看到的并非一个文件的实体，而仅仅是该文件的 inode 号和一个文件名。这是它和过去其他文件系统的最大区别所在，因为你完全可以在这个文件系统的另一处看到同样的 inode 号和一个完全不同的文件名，这种目录项与文件的关联方式被称为硬链接。在有硬链接的情况下，一个文件可以拥有多个不同的文件名和路径，但它们的内容是一样的，这体现了文件共享。在 inode 中有一个属性 i_nlinks 记录自身被硬链接的次数，如果有用户删除了一个文件，那么该文件的 inode 的 i_nlinks 将减 1，此时系统会检查它是否已经为 0，如果大于 0，则系统不会真正删除该文件，而仅仅是删除了该文件的一个目录项，因为还有其他用户拥有这个文件。只有当 i_nlinks 为 0，文件实体才会被删除。这个机制也是硬链接与软链接的最大区别之一，也是硬链接比软链接更好体现了多用户共享同一文件的有力证明。但是硬链接也确实有诸多限制，例如硬链接不能跨越文件系统，每个文件系统都有属于它自己的 inode 表，一个 inode 不能链接到另一个文件系统的文件上。硬链接不能链接目录，这是一个安全上的考虑。如果递归一个删除一个目录，而这个目录中有一个硬链接会链接到另外一个目录的话，这将是极其危险的。这些缺陷导致硬链接并没有软链接那么用途广泛。

在 Linux 0.11 系统中，有一张内存 inode 表，长 32 项，记录所有被载入了的 inode，注意存在 inode 表并不代表从磁盘载入 inode 不经过高速缓冲区。inode 的属性和缓冲头有点类似，也有等待队列，引用计数，是否锁定，是否为脏 inode，是否已更新等。但是这些属性与缓冲头的属性毫无关系，相互独立。这表示如果你想将修改过的 inode 写入磁盘，必须先等 inode 解锁，然后修改 inode 并设置为脏，等待 inode 表同步，解锁缓冲块后将 inode 写入，再等待缓冲区同步，最终将缓冲区的 inode 数据写入磁盘。这套流程必须完整，不能混淆。幸运的是，有关缓冲区的所有机制在前面的章节中已经论述，下面我们只要关注 inode 表的机制即可。

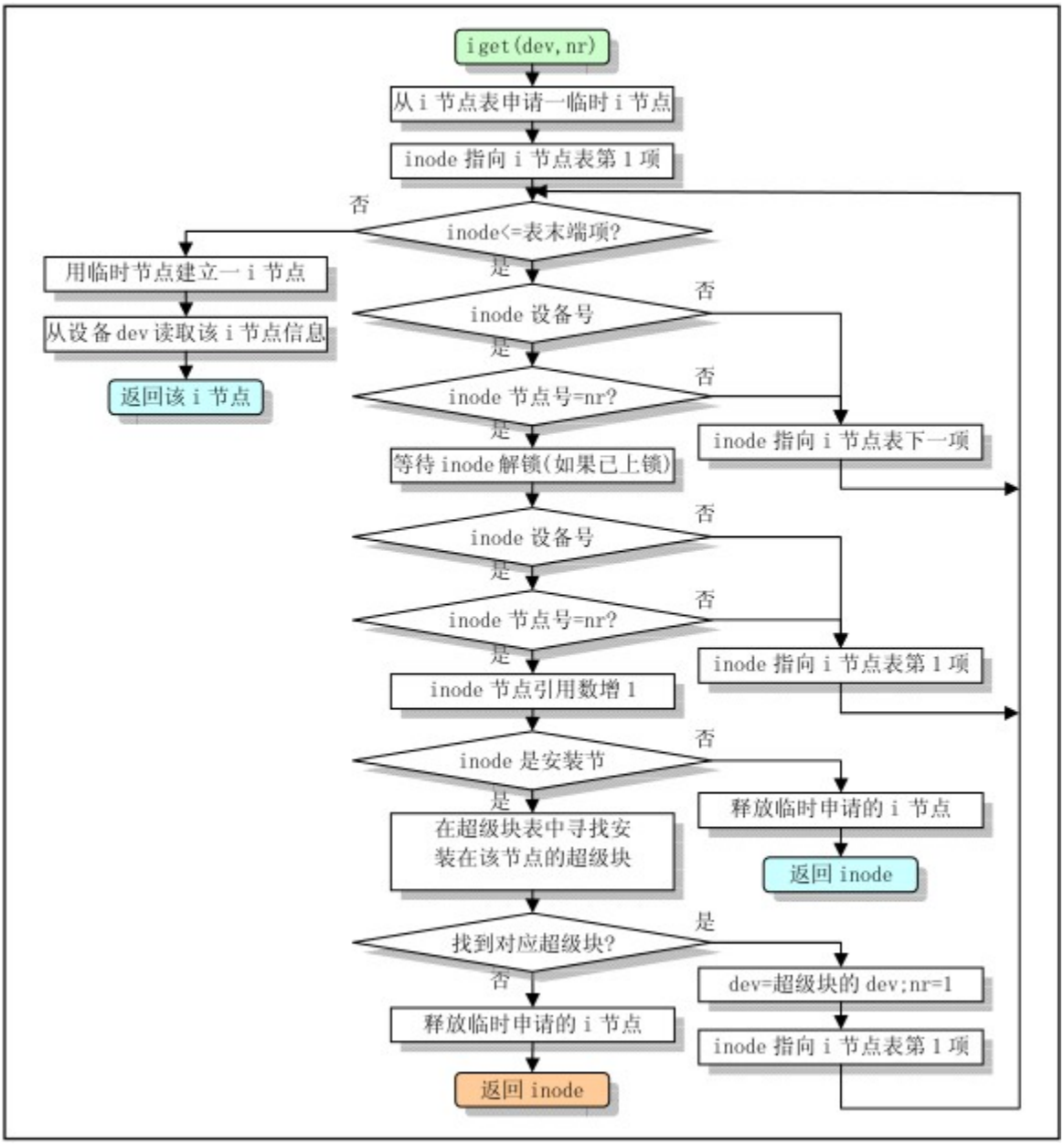
_bmap() 查询某个 inode 中记录的文件的第 block 块的逻辑块号，如果这个逻辑块号为 0，也就是说尚没有分配，且 create 参数为 1，则分配一个新的逻辑块，修改 inode，返回逻辑块号，如果 create 参数为 0，就直接返回 0。该函数虽然代码量较大，但算法很容易理解。首先检查 block 是否小于 7，如果小于，则证明 block 在直接块中，按照刚刚描述的机制返回它的逻辑块号。否则 block 自减 7，检查是否小于 512，如果小于，则证明 block 在一级间接块中，如果还没有分配一级间接块且 create 参数为 1，那么分配它。调用 bread()读入一级间接块，按照上述机制返回逻辑块号。否则 block 自减 512，用与一级间接块相类似的机制，返回逻辑块号。

get_empty_inode() 从 inode 表中获取一个空闲的（即引用计数为 0）的 inode 结点，返回指向它的指针。这个函数的算法相对于之前的搜索空闲缓冲块的算法简单了很多，但也显得粗糙了很多。首先遍历 inode 表，试图找到一个引用计数为 0，且既不锁定又不脏的 inode，如果没有，但是曾经找到过引用计数为 0 的 inode，那么也将就了。但如果连一个引用计数为 0 的 inode 都不曾发现

过，Kernel 报错死机。（有必要么？）然后等待 inode 解锁，如果是 inode，还要睡眠等待写入。由于睡眠期间可能再次造成 inode 变脏，所以要循环判断。由于睡眠也会引起 inode 被其他进程占用，还要判断是否引用计数大于 0，如果大于 0，将前功尽弃，重新搜索。最后将找到的 inode 清 0，设置引用计数为 1，返回指针。

iget() 将指定设备号的指定 inode 载入 inode 表，并返回它的地址。首先它先申请到 inode 表上的一个空位，然后开始遍历 inode 表，以期待找到一个设备号和 inode 号均符合要求的 inode，如果找到，说明 inode 已经被载入了内存。然后等待这个 inode 解锁，由于等待解锁需要睡眠，这期间这个 inode 可能被其他进程占用，所以需要重新判断它的设备号和 inode 号，如果发现改变就重新遍历。增加 inode 的引用计数，判断他是否是某一分区挂载的目录，如果不是，释放掉之前申请的 inode 空位，并且返回指向这个 inode 的指针。如果是，这个函数对此有一个特殊处理，他将返回被挂载的这个分区的根目录，即找到这个分区的文件系统的设备号和 inode 号为 1 的 inode 然后返回。因此他改变了要搜索的设备号和 inode 号，重新搜索。如果找不到，证明这个 inode 没有被载入 inode 表，那么就是用前面获取的 inode 表的空位，对他进行设置，然后返回指向这个 inode 的指针。

我不能理解的是为什么他要先申请好空位再进行搜索，而不是已经确定找不到再申请。因为申请需要一定的时间，并且会淘汰一个已经载入的 inode，严重降低了效率，不划算。如果这个空位还申请不到，那就更糟糕了。因为 inode 表中可能已经存在匹配项，但是由于申请不到空位，get_empty_inode()可能导致死机，而其实原本可以不必申请的。



iput() 放回一个被载入进 inode 表的 inode 结点。首先，等待 inode 解锁，如果 inode 是管道，唤醒等待该管道的进程，引用计数减 1，如果引用计数大于 0，返回，否则清空 inode 的占用的内存页面，并将 inode 所有属性复位。如果 inode 的设备号为 0，引用计数减 1 后返回，如果是块设备，写回这个设备所有脏块，由于写回需要睡眠，睡眠期间可能被锁住，因此需要再次解锁。如果引用计数大于 1，减 1 后返回。如果 inode 被硬链接次数等于 0，彻底删除 inode 及其文件的逻辑块，返回。如果 inode 为脏，写回 inode，写回需要睡眠，因此需要重新判断 inode 的引用计数，被硬连接次数和是否是脏块。最后引用计数减 1 后返回。

read_inode() 提供一个含有基本信息的 inode 指针，根据这些基本信息从磁盘中读入含有指定的

inode 结点的盘块到高速缓冲区，将指定的 inode 数据赋值给 inode 指针的目标。首先先锁定 inode，通过 inode 中记录的设备号找到设备的超级块，使用公式（1（指引导块）+1（指超级块）+inode 位图占用的块数+逻辑块位图占用的块数+inode 号-1）/16（每个盘块块含有的 inode 数量）获取含有指定的 inode 结点的盘块的盘块号，使用 bread 函数将盘块读入高速缓冲区，将缓冲块中的含有 inode 读出，复制给 inode 指针指向的 inode，最后释放缓冲块，解锁 inode 即可。

write_inode() 将提供的 inode 指针指向的 inode 信息写入缓冲区。无论是逻辑还是公式都与 read_inode()相似，惟一的区别在于是将 inode 指针指向的 inode 信息写入缓冲区中的 inode，然后将缓冲区设置为脏块，inode 设置为干净，即可。

第七节. 超级块

Linux 0.11 对超级块的管理与对 inode 的管理是很相似的。它在内存中定义了一个长度为 8 的超级块数组，这说明 Linux 0.11 只支持最多 8 个文件系统同时挂载。另外 Linux 0.11 只支持一种文件系统，即 Minix 1.0 文件系统，其余文件系统一律不予支持。判断文件系统的方法是判断它的魔数。每个文件系统都有一个指定的魔数，写在文件系统超级块中，Minix 1.0 的文件系统的魔数是 0x137f。

read_super() 将在超级块表中搜索指定设备的超级块，如果已经存在，返回指向超级块的指针。如果不存在，现在超级块表中找一空项，如果没有空项，返回。找到空项后，对其逐项设置，然后锁定超级块，调用 bread(dev,1)获取指定设备 dev 的超级块，判定它的文件系统是否是 Minix 1.0 文件系统，然后初始化超级块的两张位图表，然后调用 bread 从磁盘中读入两张位图表，最后做一次判定，看看读入的位图表的块数是否与超级块中记录的一致，如果不一致，释放之前读入的缓冲块。如果一致，将两张位图的第一个位均设置为 1，解锁超级块，返回指向超级块的指针。

sys_mount() 是挂载文件系统的系统调用，与 Bash 中的 mount 命令在格式上也很相似。首先根据提供的设备文件路径获取 inode 结点，然后获取记录在 inode 中的设备号，判定是否是块设备，因为只有块设备才可以挂载。put 获取的 inode 结点。检查要挂载的目录的路径，获取它的 inode 号，确定是否合法，是否引用计数为 0，是否已经是根目录，是否确实是目录。如果有一项不满足，put 获取的 inode 结点，出错返回。读取超级块送入超级块表，如果失败（可能由于超级块表满），put 获取的 inode 结点，出错返回。然后检查超级块，确定他是否当前已经被挂载，再检查要挂载的目录是否已经挂载了某个文件系统，如果是，同样 put 获取的 inode 结点，出错返回。最后设置超级块的挂载属性为目录 inode 号，设置目录的挂载标记，设置目录的 inode 为脏。最后成功返回。

可以注意到参数中的读写标志根本没有被使用过。

sys_umount() 卸载文件系统的系统调用。在 Bash 中，umount 命令的参数既可以是设备文件，也可以是被挂载的目录，不过在这里参数只有设备文件。使用和 sys_mount()完全一致的方法获取到设备号，从超级块表中找到设备号对应的超级块，检查超级块是否被挂载，检查超级块指示的挂载目录的 inode 是否也被设置为挂载。全部满足后检查 inode 表中的挂载目录对应的 inode 的引用计数是否为 0，如果不为 0 则出错返回。最后复位挂载目录的挂载标记，put 目录的 inode，复位超级块的挂载标记，put 超级块对应的 inode，复位指向超级块的根目录指针，从超级块表中删除超级块，写回设备的数据。

mount_root() 在系统开机时初始化文件表数组和超级块数组，挂载根目录，设置当前工作目录和根目录的 inode，最后统计根文件系统上空闲的逻辑块数和 inode 数，并输出这些信息。

第八节. 文件路径

Minix 1.0 文件系统的目录每项只有两个数据，2 个字节的 inode 号，和最长 14 个字节的文件名（如果超过，在 Linux 0.11 系统中可以用宏定义来选择是截断还是报错）。与 FAT 文件系统不同，目录不会记录文件属性或是逻辑块号，因为他们都在 inode 中被记录，只要知道 inode 号，就可以找到 inode 获取 inode 信息，这样显得更容易管理。

和大多数文件系统一样，一个目录中总是有.和..两项，含义也完全一致，毋须重复，我们只需要知道对这两项 Kernel 必须做一些特别处理，所以要额外关注。

另外对于例如/usr//bin 这样的写法，中间连续两个斜杠将被视为一个斜杠，必须对此做特殊处理。

find_entry() 在目录中按文件名搜索，返回指向含有目录项的缓冲块的指针和指向目录项的指针。首先根据目录的 inode 记录的文件大小和一个目录项的大小（16 字节）计算出目录中的项数。然后处理当前进程可能被设置的 chroot 机制（我没有系统学习过 Linux/Unix 编程，对此不太了解）。如果当前工作目录是一个文件系统的根目录，而该函数搜索的是..的话，将 put 并且重新设置需要查找的目录的 inode，并使它的引用计数加 1（如果当前工作目录并非一个文件系统的根目录并且搜索的是..的话，将没有必要特殊处理）。最后，逐项搜索目录的目录项，并且只在一个目录的逻辑块被查找完毕后才载入下一个逻辑块，而不是一下子全部载入。如果找到匹配项，立即返回。

get_dir() 从当前工作目录开始跟踪路径，返回路径最终的目录项所属目录的 inode 指针。首先判断是否路径第一个字符为/，如果是，将识别为绝对路径，put 当前工作目录的 inode 并从根文件系统开始搜索（Linux 0.11 似乎不支持~字符，即用户家目录）。以后，循环处理每一个用斜杠分割的目录项，如果找不到就返回，如果找到，保存 inode 指针，如果跟踪到一个非目录文件的目录项并且它还不是最终目录项，或是没有进入目录的权限，出错返回。如果跟踪到最终目录项，或是路

径的最后一个字符为/，则返回 inode 指针。

namei() 相当于基于 get_dir()继续完成通过路径最终文件和目录的 inode 指针的任务。首先找到最终文件的所属目录的 inode 指针，如果路径的最后一个字符是/，直接返回 inode 指针，否则使用 find_entry()函数找到最终文件，获取对应的 inode 指针，修改 inode 的最近访问时间，然后返回 inode 指针。

open_namei() 功能类似于 namei()，但是引入了打开属性，使他有比 namei()更强的权限判定和更多的行为。如果路径最后一个字符为/，或是最终目录项是一个目录，但打开模式却有 O_ACCMODE 或 O_CREAT 或 O_TRUNC 存在，证明用户要求打开的是文件而不是目录，那么出错返回。如果对目录没有访问权，也出错返回。如果找不到路径的最终目录项，但访问模式中拥有 O_CREAT 而且权限中拥有 MAY_WRITE，则他会检查目录项所属的目录的写权限。如果满足，创建一个空文件并添加进目录项后返回。如果打开模式中有 O_TRUNC，将删除文件的所有逻辑块后再返回。

sys_mkdir() 创建一个新目录的系统调用。首先搜索到最终目录项的所属目录，判定是否对他拥有写入权，满足后查找是否已有同名目录项，如果没有，创建一个新的 inode，设置它的 size 为 32，即拥有两个目录项。设置 inode 为脏并且修改最近修改时间和最近访问时间为当前时间。为 inode 创建第一个盘块，将盘块读入缓冲区，设置它的第一项的 inode 号为新目录的 inode 号，文件名为.，设置它的第二项的 inode 号为新目录的父目录的 inode 号，它的文件名为..，设置这个缓冲块为脏块，属性为目录且默认权限为(指定权限 & 0777 & ~current->umask)。将新目录添加新父目录的目录项，设置目录项的 inode。设置父目录的 inode，使其引用计数加 1，为脏 inode。

sys_rmdir()，sys_link()，sys_unlink()，很常用的系统调用，功能人尽皆知，而且代码非常简单，类似的机制在之前都已经出现过，因此不再详细叙述。只指出删除一个目录项就是将这个目录项的 inode 设置为 0，如果我们能够得知一个被删除的文件的 inode，我们也可以尝试用相反的手段将文件恢复。

第九节. 块设备文件的读写

虽然第一次接触，但其实对这种设备已经完全没有陌生感，磁盘就是典型的块设备。可以随机访问，只能以盘块为最小单位读写。在 Linux 0.11 系统中，内存与块设备永远没有直接交互，二者都只与高速缓冲区进行数据传输。

块设备的读写函数在 fs/block_dev.c 中定义，提供要读写的设备的设备号，给定了偏移量，要求读出指定长度的数据到内存空间或写入指定长度的数据到缓冲区，长度都可以超过一个盘块的大小，

但不能超过设备总大小。

block_read()与 block_write()的运作机制非常类似，这里只叙述 block_write()。

block_write() 负责向指定设备的指定偏移量写入指定长度的指定数据到缓冲区。函数执行一个循环，计算当前要写入的字符数，公式是 BLOCK 的大小减去指定偏移量，如果这个值大于数据的长度，那么以数据长度为结果。如果正好只写一块，将指定块读入缓冲区。否则不仅要这么做，还要预读取后两块数据。然后将指定数据写入缓冲区，指向数据的指针将后移，已经写入的数据长度将增加，还需要写入的数据长度将减少，将偏移量设为 0（因为偏移量只是针对第一个盘块），如果要写入的数据长度为 0，循环结束，返回已经写入的数据长度。

第十节. 字符设备文件的读写

字符设备在 Linux 0.11 系统中还没有完全实现，除了 tty 以外，很多函数根本不做任何有意义的事情。因此研究价值不大，从未完工的代码来看，调用字符设备的主要函数是 rw_char()，传入设备号，获得主设备号，然后以此为索引，在 crw_table 中找到相应设备的回调函数回调即可。

第十一节. 普通文件的读写

与对块设备文件的读写非常类似，fs/file_dev.c 的代码实现了对普通文件的读写。提供文件的 inode 指针，要求文件的 filp 结构（这表示文件必须已经打开，这样就可以获得文件当前的读写指针，作为偏移量，与块设备文件的读写不同，你不能随意指定偏移量），要求内存空间和数据长度。

file_write() 首先判定文件打开模式（通过 filp 获得）是否有 O_APPEND，如果有，将文件指针移动到文件尾部。执行一个循环，获取要写入的盘块的缓冲块，如果没有这个盘块，则创建一个新盘块。设置盘块为脏块，计算要写入的偏移量，移动指针到盘块的偏移量位置，计算要写入的数据长度，不超过一个盘块的范围，也不超过长度要求。计算写入以后文件的大小，如果超过当前的文件大小，则应该修改文件大小，并设置文件的 inode 为脏。然后将数据写入缓冲区，已经写入的数据长度将增加，如果已经写入的数据长度达到要求，结束循环，设置文件的最近修改时间为当前时间，如果打开模式不是 O_APPEND，还需要调整文件读写指针。设置 inode 最近修改时间为当前时间。最后返回已经写入了的数据长度。

file_read() 同样执行一个循环，他读取文件指针指向的逻辑块，计算要读取的数据长度，同样不超过一个盘块的范围和长度要求。先后移文件读写指针，使还没有读取的数据长度减少，然后进行内存复制。如果还没有读取的数据长度已经归 0，则退出循环，设置 inode 的最近访问时间为当前时间。返回读取的数据长度。

第十四节. 文件系统的系统调用

在 Linux 系统中，read()和 write()分别是读和写文件的系统调用，他们既可以被用来读写普通文件，也可以读写目录，块设备文件，字符设备文件和管道文件，可谓全能，对于 Linux 这种系统调用不能超过三个参数的情况下，能出现这样的设计，不得不佩服这个设计者的智慧。事实上，read()和 write()本身并不直接负责多种不同类型的文件的读写操作，它们仅仅是根据文件句柄在 filp 中找到对应项，获取文件的 inode，判定它的类型。最后根据类型来调用相应的函数即可，这就是这两个系统调用的全部逻辑。而这些在背后默默工作的函数在之前的第九到第十一节中都已经描述清楚了。因此，这里不再复述。

第十三节. 文件的打开和关闭

在 Linux 0.11 系统中，每个进程的 task_struct 中都有一个名为 filp 的文件指针类型的数组，长度为 20。每当一个进程打开一个文件时，这个文件的地址将被载入该进程的 filp 数组，这也就是说，在 Linux 0.11 系统中，每个进程最多打开 20 个文件。fork()系统调用会复制父进程的 filp 数组，这样，相当于子进程和父进程打开了相同的文件。close_on_exec 位图的每一位都与 filp 数组的一项相对应，指示了当进程执行 exec 系列函数时，是否要关闭这个文件。如果置为 1 则关闭，否则保持打开状态。

Linux 0.11 系统还有一张全局的文件表，被打开的文件同样出现在这张文件表中，但是这张文件表的类型是 file 型数据结构，记录了被打开的文件的源信息。这张文件表长度 64，也就是说，Linux 0.11 最大支持 64 个文件被同时打开。

sys_open() 首先设置文件的打开模式，然后在 filp 中找到一个空位（file 指针为空），如果找不到将出错退出。然后设置当前进程的 close_on_exec 位图，将找到的 filp 空位的索引值在 close_on_exec 位图的对应位置上标记为 0。然后试图在文件表中找到一个空位（文件引用计数为 0），如果找不到也一样出错退出，如果找到，引用计数加 1，调用 open_namei()函数获取被打开的文件的 inode 指针，最后初始化文件表中文件的各个属性，返回被打开的文件的句柄号。

sys_close()首先复位进程 close_on_exec 位图的对应位，使 filp 数组中对应项指针悬垂，使文件表中对应项的引用计数减 1，如果引用计数因此为 0，将 put 这个文件的 inode。

第十四节. 二进制程序和 shell 脚本的加载和执行

Linux 0.11 系统中加载和执行一个二进制程序和 shell 脚本都是用 do_execve()函数，只要理解这个函数，就理解了 Linux 加载和执行一个二进制程序和 shell 脚本和机制。这个函数代码很长，并且涉及了许多之前没有提及过的知识，需要高度重视。

1. 需求加载机制

   需求加载这套思想在 Linux 操作系统中其实已经反复出现，习以为常了，如果还不能理解的话，我引用《More Effective C++》上的一个比喻。

   “还记得你是个小孩子的时候你的父母要你打扫自己的房间么？如果你像我一样，你会说‘好的’，然后马上回头继续做自己的事情。你根本就不想打扫自己的房间。实际上，打扫房间在你心里是最不重要的一件事情，直到你听到你父母下楼检查你的房间是否真的打扫好了。然后你会冲进自己的房间以最快的速度收拾一下。如果你运气好，你的父母可能永远不会检查你的房间，你就逃掉了原本应该做的那些打扫工作。”

   在之前讲过的 find_entry()查找目录项函数时这个思想已经体现出来了，只当目前读入的数据块中找不到要求的数据时才载入一个新的数据块，如果已经找到，后面的数据块就不再载入了。这样，查找的效率就可以提升。

   在 Linux 系统中，有意的做了一套内存缺页处理机制。当 CPU 读到一个地址，而发现含有这个地址的页表并不存在于内存中时，将自动发出一个缺页异常，这个异常会被立即捕获，并且回调一个函数，将使用一个公式：含有所需数据的盘块=内存地址/盘块大小 1K+1（+1 的含义见下文第四点），然后立即访问磁盘，将 4 个盘块的数据读入页表中，然后返回。随后 CPU 将继续从刚刚发出异常的位置继续正常执行。正因为有了这套机制，我们的内存总是用的如此之慢，4GB 的内存就能跑起 10GB 的游戏，因为只要没有需求，数据根本不会载入内存。也正因为有了这套机制，我们的程序载入也非常之快，10GB 的游戏一瞬间就能打开。

   do_execve()函数会彻底停止当前进程的继续运行。也就是说，凡是写在 do_execve()之后的代码将没有可能执行，除非 do_execve()执行出错。do_execve()函数总是会清理当前进程的所有页表（内存分配表对应项引用计数减 1，如果引用计数为 0 就彻底删除）并创建新的页表，修改 task_struct 结构和 eip，esp 令其指向新的位置，然后返回，等待进程调度。系统会载入新程序然后执行，这样就完成了程序的执行。

2. 参数和环境变量的加载

   所谓参数，是指运行程序时写在程序后面的数据，比如 ls -l -a，-l -a 就是参数。他们将被传入应用程序的 argv 数组的第一第二项中（第零项则是 ls），同时 argc 将等于 2，代表有两

个参数，应用程序读取这些数据，就能产生符合用户特定需求的行为了。

所谓环境变量，可以打开 Terminal，输入 env 命令，查看输出内容。

```
ORBIT_SOCKETDIR=/tmp/orbit-bachue
HOSTNAME=localhost.localdomain
IMSETTINGS_INTEGRATE_DESKTOP=yes
GPG_AGENT_INFO=/tmp/keyring-gNcFTm/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=298b13314cac439050f857cb00000010-1299376938.321683-1770619089
HISTSIZE=1000
WINDOWID=96471651
GNOME_KEYRING_CONTROL=/tmp/keyring-gNcFTm
IMSETTINGS_MODULE=FCITX
USER=bachue
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or
=40;31;01:mi=01;05;37;41:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.ta
r=01;31:*.tgz=01;31:*.arj=01;31:*.taz=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31
:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lz=01;31:*.xz=01;31:*.bz2=01;31:*.t
bz=01;31:*.tbz2=01;31:*.bz=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:
*.ear=01;31:*.sar=01;31:*.rar=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;
31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tg
a=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;3
5:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.ogm
=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*
.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;3
5:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.axv=01;35:*.anx=0
1;35:*.ogv=01;35:*.ogx=01;35:*.aac=01;36:*.au=01;36:*.flac=01;36:*.mid=01;36:*.midi=01;36:*.
mka=01;36:*.mp3=01;36:*.mpc=01;36:*.ogg=01;36:*.ra=01;36:*.wav=01;36:*.axa=01;36:*.oga=01;36
:*.spx=01;36:*.xspf=01;36:
SSH_AUTH_SOCK=/tmp/keyring-gNcFTm/ssh
SESSION_MANAGER=local/unix:@/tmp/.ICE-unix/2044,unix/unix:/tmp/.ICE-unix/2044
USERNAME=bachue
DESKTOP_SESSION=gnome
MAIL=/var/spool/mail/bachue
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/bachue/bin
QT_IM_MODULE=xim
PWD=/home/bachue
XMODIFIERS=@im=fcitx
GDM_KEYBOARD_LAYOUT=us
LANG=en_US.utf8
GNOME_KEYRING_PID=2036
GDM_LANG=en_US.utf8
GDMSESSION=gnome
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
HISTCONTROL=ignoredups
```

```
HOME=/home/bachue
SHLVL=2
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=bachue
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
3zzn2hF02F,guid=c61d8e5a88422d188afd007100000042
LESSOPEN=||/usr/bin/lesspipe.sh %s
WINDOWPATH=1
DISPLAY=:0.0
G_BROKEN_FILENAMES=1
XAUTHORITY=/var/run/gdm/auth-for-bachue-5htEzm/database
COLORTERM=gnome-terminal
_=/usr/bin/env
```

可以看到主要是当前 shell 的执行信息。

当使用这个 shell 执行程序时，shell 会将 env 传入应用程序的 envp 数组，这样应用程序也可以根据当前 shell 情况产生一些特殊的行为。

而这些机制的实现都由 do_execve()函数完成，这些参数和环境变量可以占最多 128KB 的空间，他们被放置在应用程序那 64MB 逻辑地址的最末部分，首先放入环境变量数组的数据，然后放入参数数组的数据（这些数据是以字节为单位直接复制的，不进行任何处理），然后放入环境变量指针数组和参数指针数组，注意，这些数组的每一项都是指向之前放入的数据中的一项的指针，每根指针都和一项数据对应，并且都是以空指针结束的。然后依次放入 envp——指向环境变量指针数组的指针，argv——指向参数指针数组的指针，argc 是参数数量（由 count()函数计算），他们就是我们在应用程序主函数中被传入的参数了。

3. 初步处理 shell 脚本

   do_execve()函数将试图读取可执行文件的第一个盘块中的第一行，判断他的第一第二个字符是否是#和！，如果是，将判定该文件是 shell 脚本，随后，do_execve()将读取紧接在#!后的脚本解释器的路径，以及跟随在这个路径后的解释器的参数。然后将参数用上面描述的处理方式进行处理，同时脚本文件的路径将是脚本解释器的最后一个参数（如果 do_execve()自身没有参数的话。如果有，那么这些参数会附在脚本文件的路径后），然后重新执行 do_execve()函数，执行脚本解释器，来解释运行 shell 脚本。

4. 加载二进制程序

   如果可执行文件的第一第二个字符并非#和！，将判定为二进制程序。然后使用 a.out 格式读取可执行文件的源信息，并修改页表，进程 task_struct，eip，esp，最后返回等待调度。

   这里的 a.out 执行文件格式是重点，当应用程序执行前，操作系统必须知道这个应用程序的

代码段数据段需要多大，载入方式又是什么，应用程序的第一条指令在哪里。除此以外，如果要调试运行程序，还必须从中获取必要的调试信息。这就是 a.out 的作用。Linux 0.11 只支持 a.out 这一种执行文件格式，虽然这种格式在今天已经普遍被更强大的 ELF 取代。

可以使用 objdump -h 命令观察一个执行文件的格式。

a.out 格式在 Linux 0.11 中被定义为 exec 结构体，在 include/a.out.h 中被定义。必须特别注意 exec 结构体的第一个数据 a_midmag，他是一个魔数，告之应用程序各个部分的加载方式。Linux 0.11 只支持一种方式，被称为 ZMAGIC，这是最迎合 Linux 的“需求加载机制”的一种方式，他要求应用程序各个部分必须严格按照页面来分界而不是连续存储（即使是只有 32 字节的 exec 结构体，也必须占用 1K 空间，即一个盘块的大小），这样就很容易以页表为单位对各个部分进行逐个加载。还有一个重要的数据是 a_entry，代表应用程序的第一条指令的地址，将他赋值给调用这个系统中断的进程的 eip，这样当调度到这个进程时，eip 就可以从应用程序第一条指令开始执行了。

接下来将详细叙述 do_execve()函数的行为。首先，将检测当前进程是否运行在系统态下，do_execve()会清理当前进程的所有页表，当然不能让他清理掉系统的内存，因此必须予以检查。随后加载可执行文件，计算参数和环境变量的数量并赋值给 argc 和 engc，判断可执行文件是否是普通文件（可执行文件无论是二进制程序还是脚本文件都属于普通文件），判断是否拥有可执行权，超级用户可以享受豁免。最后读取可执行文件第一个盘块，第一阶段结束。

第二阶段中二进制应用程序和 shell 脚本的行为将产生区别，判断方式之前已经说过，是判断盘块的第一第二个字符是否是#和！，满足即是 shell 脚本，不满足即是二进制应用程序。

如果是 shell 脚本：

将紧接着读入一行（最多 127 个字符）进内存，然后找出其中第一个连续字符串（由于是用户编写的脚本文件，对格式要求不能过高），这就是脚本解释程序的路径。

用一个指向应用程序所在段最后位置（64MB-4B）的指针 p，反向复制环境变量的数据，参数的数据，每复制一个字节，p 向内存低地址移动，还要继续复制脚本文件路径的字符串首地址，脚本解释程序的参数（也就是脚本第一行除去#！和 shell 脚本解释程序路径后的剩余部分），和脚本解释程序的路径（上述的这三个数据也将作为在 do_execve()函数自身参数的基础上额外加出的三个参数传入 shell 脚本解释程序主函数，因此 argc 需要加 3）。

execve()函数的环境变量的数据
execve()函数的参数数据
脚本文件路径

shell 脚本解释程序的参数
Shell 脚本解释程序的路径
...

put 脚本文件的 inode，获取脚本解释程序的 inode 来替代，然后返回之前第二阶段的起始位置，加载 inode 指向的第一个盘块，重新执行了 do_execve()程序。一般情况下，脚本解释程序不太会是另一个 shell 脚本，因此几乎总是转入对二进制应用程序的处理。

如果是二进制应用程序：

首先以 exec 结构体的格式读取二进制应用程序第一个盘块的数据，判断魔数是否是 ZMAGIC，如果不是，拒绝执行，如果二进制应用程序过大（大于 50MB）或是目前版本的 Linux 还无法执行这种应用程序，或是这个程序看上去有残缺，拒绝执行。如果代码段初始位置不在第 1024 字节上（之前已说过，虽然 exec 结构体只占 32 字节，但根据 ZMAGIC 要求，它占用一个盘块大小，代码段紧随其后），拒绝执行。（根据之前已经计算出的 engc 和 argc）反向复制环境变量的数据，参数的数据到应用程序所在段最后位置，但如果之前已经复制过一次，就不能再复制了。设置调用系统中断的进程的 task_struct 结构，put 其中记录的可执行文件的 inode 号，重新设置为要执行的文件的 inode 号，复位信号处理和信号屏蔽位，关闭所有在 close_on_exec 位中被置位的文件句柄对应的文件，释放所有页表（内存分配表对应项引用计数减 1）（但不必重新申请页表以期待缺页中断），修改 ldt 以适应新任务，将之前设置过的应用程序所在段最后位置最后 128K 数据与数据段逻辑地址最后的 128K 进行映射。然后再在它前面反向创建 envp，argv 指针数组（以 argv 为例，遍历参数数据，每当找到一个空，就把 argv 指针数组的一根指针指向后面的数据，循环这个操作共 argc 次，最后不忘记设置一根空指针表示结束），以及 envp 指针，argv 指针，argc，详细结构在前面已经论述。最后设定堆栈段起始地址令其指向 argc，然后对齐在页面边界（这里有个疑问，刚进入 create_tables()函数时，p 应该指向段表末尾-128K 参数和环境变量数据大小，而 create_tables()函数返回值则是 p-(envc+argc+5)×4，由于 envc 和 argc 未知，返回值不是常数，一旦 p 对齐页面边界，p 与之前压栈的 argc 的距离也不是常数，那么应用程序主函数该如何找到 argc 呢？）。再设置下进程拥有主和拥有组，设定进程的 eip 为 exec 结构体的 a_entry，esp 为堆栈段起始地址，返回。

本文作者强烈要求百度退出中国！